Computational Sophistication of Games Programmed by Children: A Model for Its Measurement

LINDA WERNER, UC Santa Cruz

JILL DENNER, SHANNON CAMPE, and DAVID M. TORRES, ETR

This article builds on prior work that aims to measure computational learning (CL) during middle school. Since game computational sophistication (GCS) has been used as a proxy for a student's engagement in CL we build on their model to more completely describe the relationship between different types of building blocks of computer games and GCS. In doing so, we present a single quantitative measurement for GCS. Our model, called *GCS 2.0*, has face validity for 39 games, each programmed by a pair of middle school children. We choose four of these games, two with high GCS and two with low GCS, and discuss the computational building blocks found in each game. We do this to help the reader better understand our measurement of GCS and its relationship to CL.

CCS Concepts: • **Social and professional topics** → **Computational thinking; Student assessment; K-12 education**;

Additional Key Words and Phrases: Computational sophistication, computational thinking, computational learning, game programming, pair programming, middle school, assessment, measurement

ACM Reference format:

Linda Werner, Jill Denner, Shannon Campe, and David M. Torres. 2020. Computational Sophistication of Games Programmed by Children: A Model for Its Measurement. *ACM Trans. Comput. Educ.* 20, 2, Article 12 (February 2020), 23 pages.

<https://doi.org/10.1145/3379351>

1 INTRODUCTION

There are both national and international efforts to engage children in aspects of computational thinking (CT) from their earliest ages onward. School administrators and legislators are planning for curricula and faculty to provide this teaching; however, one roadblock to widespread adoption of CT activities in school settings is the ability to measure student learning [18]. People at the forefront of this effort describe this challenge. Cooper et al. [6] state that for education at the K-12 level, we should be focusing on computational learning (CL) since it “emphasizes the central role that a computer ... can play in enhancing the learning process and improving achievement of K-12 students in STEM and other courses.” Chris Stephenson, head of computer science education programs at Google, says, “I’m not sure yet that we understand enough about how to measure computational thinking, knowledge, and learning ... even in our own discipline space ... let alone

This work was supported by the National Science Foundation (award no. 1420596).

Authors’ addresses: L. Werner, University of California, Santa Cruz, Jack Baskin School of Engineering, 1156 High Street, MS: SOE3, Santa Cruz, CA 95064; email: llwerner@ucsc.edu; J. Denner, S. Campe, and D. Torres, 100 Enterprise Drive, Suite G300, Scotts Valley, CA 95066; emails: {jill.denner, shannon.campe, david.torres}@etr.org.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2020 Copyright held by the owner/author(s).

1946-6226/2020/02-ART12

<https://doi.org/10.1145/3379351>

across . . . other disciplines . . . There are little pieces . . . but there isn't really anything comprehensive around which we can construct a real framework for assessment yet [35]."

One place where there appears to be CL occurring is in middle and high schools, where computer game design and programming is increasingly used as a way to engage students in the kinds of complex thinking we understand as embodying CL. Studies confirm that not only is computer game programming motivating for children [3, 10], but it increases programming knowledge and CT scores [12], as well as girls' perceived computer skills and perceived support for computing [8, 9]. There are widely held assumptions about the benefits of game programming and numerous reports of CL by K-12 students programming games from various studies; however, these studies involve a wide range of age groups, programming tools, pedagogical approaches, amounts of time spent on game programming, and different ways of defining and measuring different outcomes. A recent synthesis found that despite the methodological inconsistencies across studies, there is promising evidence that game programming promotes both computational knowledge and skill. The synthesis also concluded that few measure CL, and if it is measured, these measurements differ in their outcomes [11]. Since game computational sophistication (GCS) has been used as a proxy for a student's engagement in CL [38], we build on their model to more completely describe the relationship between different types of building blocks of computer games and GCS.

In an effort to provide additional evidence about CL from game programming and details about the methods used, we organize this work around the following question:

- How can we improve the GCS model [38] to make it easier to use for assessing the CL of middle school students programming games?

We discuss an answer to this question with results from our study of middle school children using pair programming while designing and programming games. Our key contribution is a "new" version of GCS, called *GCS 2.0*, where we summarize a game's computational sophistication from the original GCS model into a single value. We believe that a single value will make it easier to assess student work. We also discuss potential uses of our model as part of an assessment framework for measuring CL. These uses include the following:

- a way for instructors to identify those students who excel and those who struggle with aspects of CL that are useful for computer game creation,
- a self-evaluation framework for students to check boxes when evaluating their own work during game creation, and
- a model for researchers to continue to validate and to build upon.

2 MEASURING CL

Researchers using Papert's constructionist approach [32] involve children in projects where they create computing artifacts such as e-textiles, animated stories, and computer games. Creating assessments for CL from these and similar projects is challenging because of cumbersome and/or manual procedures. We have noted several examples of efforts to create assessments [1, 15, 16, 20–22, 28, 39].

Brennan and Resnick [4] investigated children's projects using Scratch and have developed a CT assessment framework that looks at more than just programming constructs found in the students' projects. Their framework involves "project portfolio analysis, artifact-based interviews, and design scenarios" and is built on three dimensions of what is seen in students who engage in design activities using Scratch: the CT concepts (built in programming structures in programming language, e.g., iteration, sequence, operators, conditions), the CT practices (how the learning occurs, e.g., testing and debugging, reusing and remixing), and the CT perspectives (student shift

in perspectives, e.g., expressing, connecting, questioning). Grover et al. [17] also describe using a comprehensive assessment framework for middle school Scratch programming classes. Their studies address the design and coding of computer programs but do not explicitly address when the program being created is a computer game with one or more core game mechanics.

Specifically looking at games created by children, many researchers [5, 13, 31, 36, 39, 40] have analyzed programming instructions used in a game's source code to determine student learning. Researchers using these game analysis methods state that students who program these games show evidence of learning programming concepts. However, the analyses of the games looked only at the low-level programming instructions, similar to the research with Scratch programs mentioned earlier, are not easily scaled up for larger numbers of students, nor are they easily transferable to projects or programming languages other than those for which the analysis methods are designed.

A more comprehensive model for measuring CL from building games, the GCS model [38], is based on the analysis of 231 games programmed by 325 11 to 12 year olds using the Alice programming environment. In addition to low-level programming constructs, this model looks at the programming patterns and game mechanics that have been added by the students to their computer games' source code files. These authors aligned their model with computer science education research-based work by Linn [26] and Robins et al. [33] that uses a programming framework to describe how novices learn to program. To emphasize this alignment, the framework is described next with the corresponding dimensions of the GCS model indicated in parentheses:

- (1) *knowledge* of design, language, and debugging tools, such as the knowledge of the language primitives (knowledge of the *programming* language low-level *constructs*);
- (2) *strategies* for design, implementing the program using a programming language, and debugging, such as the “repertoire of templates” and the “procedural skills” necessary to combine the language primitives together to solve specific problems (an understanding of how these programming constructs can be combined into *programming patterns* to provide additional program functionality); and
- (3) mental *models* of the problem domain, the desired program, and the actual program, such as the ability to plan, text, and reformulate for other problem spaces based on the mental model of the problem solution (combining programming constructs with programming patterns to produce *games mechanics* used to model the game problem space).

By representing the alignment in this way, these three GCS model dimensions also provide a useful framework for identifying the types of thinking and learning that a student is engaged in when participating in an open-ended complex problem-solving activity such as creating a computer game with a partner. Unfortunately, the three dimensions of the GCS model [38] had not been integrated into a single quantitative measurement for ease of use for assessment of student work.

GCS 2.0—our model for GCS—builds on the aforementioned GCS model [38] with its three sets of structures (i.e., programming constructs, programming patterns, and game mechanics) and consolidates sophistication into a single quantitative measurement. We have incorporated sophisticated programming constructs into the set of programming patterns so that GCS 2.0 has only two sets of structures (programming patterns and game mechanics). We discuss these changes to the original GCS model, why we made the changes, and how we analyzed games using GCS 2.0.

3 PARTICIPANTS AND PROCEDURES

Participants were in a voluntary after-school program where they worked in pairs to design and program a 3D animated game in Alice, a block-based programming environment [7]. The program ran once a week for approximately 2 hours each session over the course of 12 weeks (total

of approximately 24 hours). Activities included learning about and being supported in pair programming (e.g., video, poster, role play, reflection on roles) [12]; learning how to code in the Alice environment using a Use-Modify-Create three-stage progression [24] to engage the student with CL using self-paced exercises we call *challenges* [36] and that can be used as a formative assessment of programming knowledge; designing their game idea on paper; and then programming, testing, and revising their game in Alice. Game requirements included player interactivity, clear instructions, a clear objective for the player, and an outcome at the end of the game. Students initially worked with two or three different partners in the first two sessions and then were asked to write down their partner preferences. The teachers then paired students based on their choice, as much as possible.

The data described in this article are from 39 pairs in middle school at four schools in the Western United States; the socioeconomic levels varied—two schools had 33% eligible for free and reduced lunch, and two had 90% eligible. Of these 78 students (50% girls, 50% boys, 44.9% Latino/a, 25.6% white), 61.5% were sixth graders, 26.9% were seventh graders, and the rest were eighth graders. Students self-reported gender, and students self-reported their grades, saying that many get mostly A's and B's (49.4%), whereas 27.3% get all A's. We collected 39 games from the middle school students at the four schools.

4 GAME ANALYSIS

Two undergraduate researchers, one a computer science major and one a cognitive science major, were trained, and then analyzed 39 games for programming constructs, patterns, and game mechanics based on the original GCS model. Our researchers looked for reachable (i.e., on a path that is executable by some sequence of user game play) instances of programming constructs and patterns. This pattern analysis strategy focuses on the presence and the implementation completeness of the pattern. As seen in Table 1, the sophistication level of a pattern is labeled as 1, 2, or 3; however, a game may have a level 2 or 3 pattern but the implementation may be missing some parts. If so, depending on the completeness of the pattern, the level is coded as less than the label of that pattern. For game mechanics, researchers also based their analysis on completeness, as well as appearance through graphics (character selection, character programming, or background selection), text, and audio information found in the playable version of the game and in the program code. See the appendix for details of pattern and mechanics analysis.

Both undergraduate researchers analyzed each game looking for programming constructs. For the first 10%, the discrepancies were discussed in meetings of the two undergraduates and the senior researcher, and agreement was reached. This gave the researchers a greater shared understanding of coding categories. At this point, all of the games were independently double-coded by the undergraduate researchers who would compare and discuss any discrepancies (including obtaining guidance from a senior researcher as needed) and determine the final result. The patterns and mechanics in each game were also coded in a similar way, including discrepancies for every code being discussed and resolved. Inter-rater reliability is 97.84% for the programming constructs and 91.81% for the patterns and mechanics.

Our approach to issues of validity followed that used for the original GCS model, which is based on methods that suggest that the focus be on credibility, transferability, defendability, and confirmability [25] and on trustworthiness of our data analysis approach [30]. These include minimizing researcher bias by including multiple coders that bring a range of perspectives (gender, race/ethnicity, and prior gaming experience) and then valuing and listening to each perspective equally. We used the strategy of peer debriefing; we also externally evaluated the data analysis (based in logic rather than empirical evidence). The first paper author reviewed all of the conclusions and questioned the procedures for establishing agreement. The original GCS model lists the

Table 1. Patterns for GCS 2.0

Pattern	Level	Pattern Description
Sound	1	Use of sounds not built into Alice methods.
Key/Mouse Ctrl Movement ^{Change}	1	Controlling object or camera movement with a key or mouse.
Manipulating Subparts	1	Programming subparts of an object to change during the game (e.g., arm of one character hits another and just the character's head falls off).
Instructions ^{Change}	1	Instructions are programmed via character communication or print statements.
Billboard ^{Add}	1	Instructions are programmed via an imported billboard object.
Phantom Object	1	Using a not-in-view object to move and position other objects.
Embedded Method	1	Student-created method that is embedded within another method.
User-Usability ^{Add}	1	Instructions provided for most functionality.
Programmer-Usability ^{Add}	1	Renaming of objects with other than default names.
Parameters ^{Add}	1	Setting parameters on optional or student-created methods.
Nested If/Else ^{Add}	1	Nested if/else statements.
Repetition ^{Add}	1	Loop, while, or event found for "while something is true."
Dialog Box	2	Player is asked for input, input is read in, and program uses the input.
Vehicle	2	Vehicle property is used so that when the vehicle object moves, an attached object moves in unison with it.
Collision	2	A program action depends on the distance one object is from another.
Camera Control	2	Changing the view according to movement or player input within one scene.
Scene Change	2	Programming movement to and from different scenes.
List Processing ^{Change}	2	List variables are created and used with "For all in order," "For all together," "For all one at a time," or "For all every together."
Counter	3	Integer variable created and initialized, variable's value incremented or decremented based on game state, and if variable hits threshold value, it triggers additional action.
Timer	3	Integer variable created and initialized, variable's value changed as time passes, and if variable hits threshold value, it triggers additional action.
Expand Obj Method ^{Add}	3	Student-created method to give additional functionality to an object.

Note: The superscript "Add" indicates that the pattern is added to the original GCS model. The superscript "Change" indicates that the pattern is changed from the original GCS model.

following game characteristics that were associated with the GCS of the case studies reported in that paper:

- (1) Whether sophisticated constructs are used
- (2) Whether sophisticated and operational patterns are created
- (3) Whether the constructs and patterns are integrated into mechanics
- (4) Whether the mechanics are operational
- (5) Whether two or more mechanics are integrated [38].

Our GCS 2.0 model builds upon these five game characteristics to produce a way to create a single measurement for GCS. This new model also focuses on sophisticated programming constructs and downplays the impact of unsophisticated and commonly used programming constructs (e.g., simple event handlers that in Alice are commonly used by all students) on a game's computational sophistication. Many of the differences in programming constructs in the 39 analyzed games were for programming constructs that are already found in descriptions of the patterns (e.g., set, built-in functions, student created variables, some sophisticated event handlers, student created methods, list processing constructs) and therefore would be accounted for when analyzing for patterns. Other programming constructs from the original GCS model were actually composed of or parts of segments of code or code templates (e.g., some sophisticated event handlers, loops, while, student created parameters, nested if/else) and also would be accounted for when analyzing for patterns.

All changes to the set of patterns for GCS 2.0 are described in Section 4.1 and labeled in Table 1. Minor additions to the set of mechanics for GCS 2.0 are described in Section 4.2 and Table 2. Based on these changes, we re-analyzed the 39 games using the lists of patterns and mechanics of GCS 2.0. Since our model relies on programming constructs by their presence in the expanded set of patterns, the programming constructs are counted once when computing the GCS 2.0 score. This way, GCS 2.0 still maps to the three dimensions of the GCS model of programming knowledge (programming constructs), strategies (patterns), and mental models (mechanics). GCS 2.0, relying on all five of the preceding game characteristics, is composed of a set patterns that is expanded from that in the original GCS model (Table 1), a set of game mechanics (Table 2), and an algorithm to calculate a GCS score. See Section 4.3 for details of the algorithm.

4.1 Patterns

A programming pattern is a segment of code or code templates that may or may not be contiguous and that creates additional program functionality. To create program functionality from a template, programmers add code representing the details of the programming problem into placeholders in the template. Our set of patterns builds on the set of patterns of the original GCS model but is augmented by the addition of seven patterns and changes to three others.

We used the constant comparative method [29, 37] to determine how the set of programming constructs from the GCS model are used in several games. Then we reviewed additional games and added patterns and changed patterns as appropriate so that we see the set of patterns as encompassing all of the sophisticated programming constructs, as well as any additional patterns found in this group of games. We determined that most of the level 1 programming constructs were found in most of the games. Therefore, we did not add to the set of patterns for these items. Our changes to the set of patterns (see Table 1) involve either explicitly adding sophisticated programming constructs to existing patterns because they involve segments of code or code templates and fit the definition of pattern (e.g., list processing, key/mouse control of movement), spawning off a new pattern from an existing pattern (e.g., the billboard pattern from the instructions pattern), or adding patterns (e.g., programmer usability, user/(player) usability, expanding object method that represents aspects of object-oriented programming, nested if/else, parameters, and repetition

Table 2. Game Mechanics for GCS 2.0

Mechanic	Mechanic Description
Avoidance	Player moves an object to avoid either a stationary or moving obstacle based on player proximity to the obstacle. Feedback to proximity is required.
Collecting	Player attempts to accumulate objects to advance in a game.
Exploration	Player moves an object or the camera to find objects beyond the payer's initial range of view. Movement is not restricted to occur along a designated path.
Guessing	Player answers questions via clicking, typing, or moving an object.
Hidden Objects	Player searches for an object that is hidden either beyond view or "hidden in plain sight."
Hit Moving Target	Player attempts to click on a moving object or moves something (character, object, camera) closer to a moving object to prompt another action.
Infinite Game Play ^{Add}	Intentional for a game to have no end.
Interaction ^{Add}	Player can interact with a game.
Levels	Player moves between at least two stages by gathering points or fulfilling a challenge.
Navigation	Player moves an object and/or camera from one location to another known location often on a designated path.
Puzzles	Player organizes objects to make order out of disorder.
Racing	Player moves an object across a finish line within the time limit or moves an object in competition with other objects.
Shooting	Player shoots at an object; actual projectile must be present.
Timed Challenge	Player is given a time limit to complete game task.

The superscript "Add" indicates that the mechanic is added to the original GCS model.

for patterns created directly from some of the more sophisticated programming constructs). These changes and additions to the set of patterns for GCS 2.0 represent the dependence of patterns on the level 2, 3, and 4 programming constructs of the original GCS model. These changes were made by the primary author after consultation with two of the authors who were active working with these students and their games or analyzing student games. As seen in Table 1, the sophistication of a pattern is indicated by its designation as level 1, 2, or 3. We use an arithmetic progression in designating the levels of patterns to loosely correspond to increasing numbers of non-contiguous segments of code involved in implementing a pattern, as well as the level of sophistication of the programming constructs required. For example, most level 1 patterns in their simplest implementations are contiguous segments of code. Implementing a timer or a counter pattern requires three or four non-contiguous segments of code. In addition, the implementation of a pattern determines whether the instance of that pattern is operational and is also used in determining its level in the calculation for the GCS score. See the appendix for the details for pattern analysis and determination of a pattern's level.

4.2 Game Mechanics

Game mechanics are programmed from a combination of one or more programming constructs and zero or more programming patterns. They provide the basis for game play and provide implementations for game rules, game behavior, and game feedback [19]. It is the game mechanics that

make games fun to play and often make or break the success of a game if programming for the commercial market. We base our set of game mechanics on that found in the original GCS model [38] with the addition of two mechanics: infinite game play and interaction. We added “infinite play” to the set of game mechanics because we found this mechanic in one of the 39 student games; we added “interaction” because this mechanic represents a behavior that is necessary but not sufficient for a computer program to be considered a game. We added these two game mechanics using the constant comparative method described earlier in Section 4.1.

Just as it is important to distinguish between operational and non-operational programming patterns, it is important to distinguish between operational and non-operational mechanics. The player of a game must be able to be actively involved in a mechanic for it to be counted as operational. The rules of the game, the behavior required by the mechanic to be operational, and the level of feedback are often different for each type of mechanic. Sometimes determining a game’s mechanic requires distinguishing between a game’s underlying goal and the mechanics present to provide player interaction to meet that goal. For example, if a player is given an instruction to “answer questions to avoid being shot,” avoidance is seen as the underlying goal of the game; however, only guessing is counted as a mechanic. In addition, sometimes a mechanic is alluded to in a game’s instructions but is not implemented. This shows a student’s intent to program a certain mechanic and is enough to be counted as at least a non-operational mechanic. See the appendix.

4.3 A Single Measurement for GCS: GCS 2.0

In GCS 2.0, a GCS score is calculated from the following game metrics:

- The number of different types of patterns used in the game including their sophistication and their operational status, where the sophistication and operational status is represented by its level of either 1, 2, or 3, where 3 is the most sophisticated and operational.
- The number of different types of mechanics used in the game including their operational status.

More specifically, the algorithm is as follows:

GCS Score = Pattern Score + Mechanic Score, where

$$\begin{aligned} \text{Pattern Score} = & 3 \times (\# \text{ of types of pattern implementations at level 3}) + \\ & 2 \times (\# \text{ of types of pattern implementations at level 2}) + \\ & 1 \times (\# \text{ of types of pattern implementations at level 1}) \end{aligned}$$

$$\text{Mechanic Score} = (n(n+1))/2 + m, \text{ where}$$

n = number of types of operational mechanics and

m = number of types of non-operational mechanics

In all instances, a type of pattern or type of mechanic is counted only once per game and a pattern is counted at its highest level as seen in that game. For example, a hypothetical game has three instances of the counter pattern: one for counting points scored in the game, one for counting the number of stars clicked on by the player, and one for counting the number of moves made by the player. In this hypothetical case, the students programming this game were not successful in getting all of the counters completely working; both the counter for stars and the counter for moves were not incremented correctly, but the counter for points was declared, incremented appropriately, although there was no presence of a threshold value to trigger additional actions. The highest level of this type of pattern is 2; therefore, in this example, the game is coded to have one counter pattern at level 2. Our hypothetical game also has one sound pattern fully implemented where a fireworks sound occurs when the game starts. No other patterns are found in this game.

A fully operational sound pattern is a level 1 pattern. Since there are no other patterns in this hypothetical game, the pattern score is calculated as follows:

$$\text{Pattern score} = 3(0) + 2(1) + 1(1) = 3$$

The portion of the algorithm that relies on game mechanics accounts for each of the types of game mechanics and the increased computational sophistication that is seen in games that have the integration of more than one operational game mechanic. Our scoring algorithm gives a multiplicative increase to games that have multiple types of operational mechanics and represents the power of their interaction with using a quadratic progression. When using quadratic progressions, the difference between each pair of subsequent elements in the progression increases or decreases at a constant rate. In our algorithm, the differences between subsequent scores representing operational mechanics increase by 1, 2, 3, 4, and so forth. For example, the mechanic part of the score for a game with one operational mechanic is 1, for two operational mechanics it is 3, for three operational mechanics it is 6, for four operational mechanics it is 10, and for five operational mechanics (the largest number of operational mechanics we found in a game in this study) is 15. In this example, the differences between subsequent scores increase by 1, 2, 3, 4, and 5 with an increase in the differences by the constant “1.” For our hypothetical game, there are operational instances of both the navigation and guessing mechanic and a non-operational instance of the shooting mechanic. Since there are two operational mechanics and one non-operational mechanic, the mechanic score is calculated as follows:

$$\text{Mechanic score} = (2(2+1))/2 + 1 = 4$$

The resulting GCS score for this hypothetical game is calculated as the sum of the pattern score and the mechanic score as follows:

$$\text{GCS score} = 3 + 4 = 7$$

The process used to develop this algorithm started with the analysis of the 39 games and ordering them according to their GCS. We analyzed the code for both existence and implementation of patterns and mechanics. We also played the games to determine if they execute and run as expected given instructions within the game program’s code. We started this process with two games and ordered them in increasing GCS, adding one game at a time into its place in the set creating a partial ordering of the entire set of 39 games since some were identified as at the same GCS. At this point, there are no actual GCS scores assigned, only a partial ordering.

To develop the algorithm, since we believed that both patterns and mechanics were important in this ordering, we thought that using the simple ranking system that existed in our analysis of the patterns as seen in Table 1 could be used to score the patterns. We used a simple ranking system that is represented by an arithmetic progression where the score for a pattern of level 1, 2, or 3 is represented by a score of 1, 2, or 3 if the pattern is completely implemented. If parts of the pattern are not present in the code, the score is reduced correspondingly. This arithmetic progression does not represent any interaction between patterns, which seems appropriate since that is how we interpret the implementation of patterns in the students’ programs. All of the interacting parts are within each pattern, not between patterns.

For the mechanics part of the algorithm, we looked for a progression that represents interactions since the total group of mechanics used in a game determines the rules and playability of the game. As additional mechanics are added to a game, the rules interact with each other and make for a more computationally sophisticated game. This part of the algorithm needed to represent interaction, so a quadratic progression was chosen. If a game has either one, two, three, four, or five operational mechanics (the largest number of operational mechanics in any of the 39 games

was 5), then the score for the operational mechanics part would be 1, 3, 6, 10, or 15, respectively. Since the difference between each subsequent element in this progression increases at a constant rate, it is a quadratic progression.

This algorithm assigns a GCS score to each game, and these scores fit the partial ordering of the 39 games. We did look at an algorithm that uses a simpler model for the mechanics part. In this instance, we used an arithmetic progression of 1, 3, 5, 7, 9 for one, two, three, four, or five operational mechanics. When using this algorithm, GCS scores mis-ordered four games, giving them lower GCS scores than desired. It is interesting to note that for each of these mis-ordered games, the computational sophistication of the patterns part of their games was a very low percentage compared to the mechanics part.

The details for the levels of each of the patterns and for operational status of mechanics can be found in the appendix.

5 RESULTS: GCS SCORES

GCS 2.0 has face validity for 39 games, each programmed by a pair of middle school children using pair programming. Of the 39 games analyzed using GCS 2.0, the mean GCS score is 12.5 (range 4–23), where 20 games have GCS scores below the mean and 19 have GCS scores equal to or above the mean. Thus, this score captures a range of GCS.

In addition, the pattern score and the mechanic score also captured a range of sophistication. For the set of 39 games that we analyzed, the mean pattern portion of the GCS score is 7.7 (range 3–14). The mean number of patterns in a game is 6.1 (range 3–11), with the mean number of level 1 patterns at 4.6 (range 1–9), level 2 patterns at 1.3 (range 0–4), and level 3 patterns at 0.2 (range 0–2). The mean for the mechanic portion of the GCS scores for the 39 games is 3.6 (range 1–15). The mean number of mechanics in a game is 2.8 (range 1–6), with the mean number of operational mechanics at 2.1 (range 0–5) and the mean number of non-operational mechanics at 0.7 (range 0–4).

We believe that it is useful to look closer at the validity of GCS 2.0 to determine if the subscores tend to correlate or if it is common that a student can have a high score in one but not in another. For these correlations, we have used the construct scores computed from the original GCS model where the mean for the construct scores for the 39 games is 8.5 (range 3–25), where 25 games have construct scores below the mean and 14 have construct scores above the mean. The correlations between each of the subscores and the GCS 2.0 scores are as follows: pattern score: $r(39) = 0.746$, $p < .01$; mechanic score: $r(39) = 0.746$, $p < .01$; and construct score: $r(38) = 0.535$, $p < .01$. As can be seen, the pattern score, mechanic score, and construct score all correlate positively with the GCS final score.

Not all games have GCS scores that are composed of similar percentages from each of the pattern and mechanics subscores. For example, it is possible to have two games that have similar GCS scores, but one has a substantially higher pattern subscore than the other and the second has a substantially higher mechanic subscore than the first. Sometimes simple constructs and few patterns are used to create a game with many integrated game mechanics. It is important to realize that the GCS score measures both the sophistication of the patterns used and the sophistication and integration of the game mechanics used to create a game that is interactive and has a goal.

6 CASE STUDIES

To further explore the variation in GCS scores and provide a better understanding of GCS 2.0, we chose four games: two with higher than average GCS scores and two with lower than average GCS scores. For each game, we give details about how the GCS score was derived from the computational building blocks used in the game.

Fig. 1. Screenshot of the events window of a high GCS scored game.

6.1 Building Blocks Used in a High GCS Scored Game

The GCS score for this game is 21 (range 4–23)—the third highest GCS score of the games analyzed for this study. See Figure 1 for a screenshot of this game including some of the program code.

This is a game in which the player must prevent a lumberjack from reaching an axe and chopping down a tree. Over time, the pine tree collects “sunlight” (in-game currency), which the player can then use to buy acorns to throw at the lumberjack (using 5 sunlight) or make the tree grow (using 20 sunlight). As the lumberjack slowly moves toward the axe, the player can click on him to throw acorns, causing the lumberjack to grunt and take a step back (although instructions were given for the “acorn” but the action was not coded into the game). The game therefore has two mechanics that work correctly (hit moving target and interaction) and four mechanics that do not work correctly (collecting, timed challenge, shooting, and avoidance) for a mechanic score of 7. These four mechanics fail due to errors in the code (setting sunlight value to 1 rather than adding by one every second) and/or missing code (missing the acorn projectiles). Furthermore, the game has seven patterns at level 1 (counter, parameters, sound, embedded method, instructions, user-usability, and repetition), two patterns at level 2 (expand object method and collision), and one pattern at level 3 (timer) for a pattern score of 14 resulting in a GCS score of 21. The counter pattern is used in the game to display the number of seconds that have passed in the top left corner of the game window. Parameters and sound patterns were used when programming the speed at which the lumberjack walks and the grunt he makes when hit. Text instructions, presented to the player at the beginning of the game, explain the user controls (instructions and user-usability patterns), and embedded methods and loops are used in the game code to activate the timer and main scene. This game also receives a score of 1 for the expand object method pattern because the students created a method within the “tree” object that “shot an acorn” and made the lumberjack react (move back and grunt). Finally, the collision pattern was used to end the game when the lumberjack reached the axe signifying that the tree is chopped down.

6.2 Building Blocks Used in Another High GCS Scored Game

The GCS score for this game is 17 (range 4–23) and the highest GCS score of the games created at one of the schools. See Figure 2 for a screenshot of the game in play.

In this game, the player is presented with an igloo where three penguins are hiding from a Husky dog that is outside the igloo near a pond containing fish. The objective is for the player to collect and save the fish by clicking on them before the time runs out. This will stop a Husky dog from eating the fish. When clicked on, each fish is sent to a bucket inside the igloo. There is no way to win as programmed, as a “game over” message appears regardless of whether you save all 11 fish or not. The game has three working mechanics (collecting, timed challenge, and interaction)

Fig. 2. Screenshot of game in play.

Fig. 3. Screenshot showing a billboard with instructions for game play.

for a mechanic score of 6. The game has six patterns at level 1 (manipulating subparts, embedded methods, sound, user-usability, instructions, and repetition), one pattern at level 2 (expand object method), and one pattern at level 3 (timer) for a pattern score of 11 resulting in a GCS score of 17. Aspects of the game that use these patterns are the following: having the Husky dog move its head upward before it barks at the penguins, embedding methods within the main world method to begin the countdown timer, and having the penguins explain the game objective using penguin object speech bubbles.

6.3 Building Blocks Used in a Low GCS Scored Game

The GCS score for this game is 5 (range 4–23) and the lowest GCS score of the games created at one of the schools. See Figure 3 with screenshot showing a billboard with instructions for game play.

Fig. 4. Screenshot of game and event windows.

This game has no real goal. The scene begins with a cowboy and a baby standing beside a gazebo. The cowboy then asks the player “what do you want to be when you grow up?” and the player can choose from a list of options. These options are presented on a static billboard that takes up the top third of the game window. Once a player chooses a response, the baby transforms into that thing (e.g., an astronaut). The game has one functioning mechanic due to the player’s ability to change the state of the game (interaction), earning it a mechanic score of 1. The game uses both a billboard and character speech to explain the game’s functionality while also creating a new method for the baby object to allow it to shape-shift. Therefore, the game has two patterns at level 1 (user-usability and billboard), one pattern at level 2 (expand object method), and no patterns at level 3 for a pattern score of 4 resulting in a GCS score of 5.

6.4 Building Blocks Used in Another Low GCS Scored Game

The GCS score for this game is 4 (range 4–23) and the lowest GCS score of all games analyzed for this study. See Figure 4 with screenshot of game and event windows available for programmer edits.

This game has no real goal, and no directions are given. The game shows a penguin igloo hotel with various penguins and one snowman. Upon starting the game, the player is immediately presented with 3D text that reads “Congrats you Finish” before the camera pans in to the penguins and plays a short, seemingly unfinished scene. During this scene, one of the penguins walks up to another penguin and says “Hello how are you doing.” The snowman near the penguin then says “goodbye” and takes a step forward before another penguin says “my stomach hurts” and the scene ends. The game has no mechanics at level 2 and only one mechanic at level 1 (interaction) for a mechanic score of 1. The player can control one of the penguins with the arrow keys on the keyboard, but due to the lack of instructions, this can be determined by analysis of the program code or can be accidentally discovered during game play. Similarly, the player can make a dog bark by pressing the “N” key. The game has three patterns at level 1 (key/mouse control of movement, sound, and parameters) and no patterns at levels 2 or 3 for a pattern score of 3 resulting in a GCS score of 4. Parameters that were manipulated include the duration of the speech text and the speed of the dog’s bark.

6.5 Summary of Case Studies

The case studies of the building blocks used in the four games describe the patterns and mechanics of each game and include a short explanation of program code and program functionality for each pattern and mechanic. Additionally, a screenshot is given for each game to help the reader understand each of these games. Among the most noticeable differences between the high and low GCS

scored games is the lack of apparent goals for both of the low GCS scored games; no instructions were given in the lowest GCS scored game; only one mechanic, the interaction mechanic, was found in the low GCS scored games; and no level 3 patterns, such as counters and timers, were found in the low GCS scored games.

7 DISCUSSION

The focus of this article is on the development of a new version of the measurement model, GSC 2.0, rather than on identifying typical GCS scores. The measurements were done of games, each made by a dyad of students; all students were enrolled in elective, after-school classes. With elective classes, this generally means that students enter the classes with high motivation and some prior knowledge of computing.

The research described in this work describes part of an assessment for CL when games are designed and programmed using the Alice programming environment. But many other programming languages, such as Scratch and Python, are being used in similar school settings. We believe that a model similar to GCS 2.0 can be used to measure GCS if a different programming language is used. To use a model similar to ours for games programmed using a language different from Alice, the set of patterns of Table 1 and the set of mechanics of Table 2 must be translated to those for that other programming language. Additionally, the content of the appendix must be translated to correspond to that language's programming constructs and the patterns and game mechanics possible with that programming language. Some of this is already available for languages such as AgentSheets [2]. One important limitation of the use of GCS 2.0 is that it requires human efforts that can add subjectivity to the resulting scores. This model is not automated, nor do we believe that it can be entirely automated.

We have extended a model for GCS, calling our model *GCS 2.0*. The construction of this substantial addition to GCS has been the result of an attempt to understand the computational sophistication of the computational building blocks of games (programming constructs, programming patterns, and game mechanics) designed and programmed by K-12 students and the computational sophistication of interactions between game mechanics within a single game. It is true that some well-designed and fun-to-play games require a very simple coding structure to implement. We have developed a model to measure the sophistication of the computational pieces used to create a game rather than the sophistication (however, that may be measured) of the game play itself. Our measurement appears to distinguish between games having high GCS and games having low GCS.

As more examples of games designed and programmed by K-12 students become available, both those programmed using Alice but also games programmed in different languages, it is hoped that our model fits these additional games or can be tweaked to do so.

We believe that it is interesting to consider what might contribute to a student developing a game with a higher or lower GCS score. Our course curriculum provided students with self-paced instructional exercises built to provide scaffolding as students became more experienced with the programming language and development environment. These exercises exposed students to most of the programming patterns and to many of the game mechanisms listed in Tables 1 and 2. Approximately halfway through the course, student pairs then started designing and programming their game. We did not do any analyses looking at individual students and how their attributes (age, Alice knowledge, etc.) correlate with their final GCS score because the score comes from calculations performed about a game designed and created by a dyad. Ignoring the influence from the partner violates the assumption of independence. However, if we take the average of both of their individual attribute values, we found that a stronger friendship between the pair at the beginning of the course has a weak negative correlation with the GCS score ($r = -.377$, $p < .05$). Thus,

those students who reported higher levels of friendship with their partners at the beginning of the course had lower GCS scores at the end. No other significant correlations were found.

We believe that a student's familiarity with games, and therefore of the types of games that can be made, would contribute positively to them being able to develop a game with a higher GCS score. A student's familiarity with Boolean logic and abstraction would also contribute positively to them being able to achieve a high GCS score. Interviews with the students might have assisted us in learning more about this.

8 CONCLUSION

As stated earlier, one roadblock to widespread adoption of CT activities in K-12 school settings is the ability to measure student learning from game programming and similar activities, known as CL. The major contributions of our work are the following:

- (1) We present the design and development of a model, GCS 2.0, that produces a single measurement for the computational sophistication of a game.
- (2) We use GCS 2.0 for the calculation of the GCS score with the result of face validity for 39 games. Our model incorporates added sophistication when a game is programmed using multiple game mechanics to represent their interaction.
- (3) Most importantly, GCS 2.0 can be used as part of a larger assessment framework for measuring CL that occurs when K-12 students design and program games.

We do not suggest that the GCS 2.0 assessment model be used as a stand-alone end-of-course assessment of programming skills such as that of the FCS1test [14, 27] or the Israeli Nationwide Exam [41]. We believe that the use of GCS 2.0 to assess the computational sophistication of computer games designed and programmed by K-12 students can provide a fuller perspective on student programming and game design skills when used in conjunction within an assessment framework similar to that described by other researchers [4, 17, 34]. GCS 2.0 can be used as part of the project portfolio analysis of an assessment framework for CL. The GCS 2.0 part of a project portfolio analysis could contain an interview where students describe and demonstrate to their teacher the GCS of their game starting with a self-evaluation form with check boxes for patterns and mechanics filled out by the students during their game creation. During the interview, the students indicate which patterns and mechanics are present, show where these patterns and mechanics reside in the code, and play the game demonstrating aspects of the patterns and mechanics in action. We suggest that self-evaluation could be used to address an important limitation of the use of GCS 2.0 because of the subjectivity of human involvement in the algorithm used for measurement of GCS. Future work is needed to determine if this use of self-evaluation forms during student interviews is feasible for GCS measurement. In addition to student interviews, other parts of a comprehensive assessment would contain problems similar to those seen in the Fairy Assessment [39] or in the Design Scenarios [4] for Scratch.

Other than the work done with AgentSheets [2, 23], we know of no other CL assessment based on project portfolio analysis that includes an analysis specifically designed for computer games. Some other project portfolio assessments look at foundational CT concepts like algorithmic thinking looking for the presence of programming constructs for sequence, repetition, and alternation; however, we look for combinations of programming constructs that together are used by students to create programming patterns and combinations of programming constructs and patterns that are used by students to produce game mechanics. We hope to perform validation of the GCS 2.0 assessment model once more games created using Alice and other programming languages become available.

APPENDIX

INSTRUCTIONS FOR CODING SOPHISTICATION OF PATTERNS AND MECHANICS

Patterns

Sound. The use of sounds that are coded by the student.

0 = not present

1 = background or integrated sound

Key/Mouse Ctrl Movement. One or more of opposing two-, three-, and four-key assignments; mouse move object event, or let mouse move camera event either with instructions or not.

0 = not present

1 = programmed but either working or not working successfully

Manipulating Subparts. Code to manipulate subparts of object.

0 = not present

1 = present and either used or not used

Instructions. Using either character communication or using print statements.

0 = not present

1 = present and used

Billboard. Uses an imported billboard object to deliver clear instructions.

0 = not present

1 = present and used

Phantom Object. Using a not-in-view object to position another object in code. Dummy objects in Alice are not phantom objects; see the Camera Control and Scene Changes sections.

0 = not present

1 = present and used in code

Embedded Method. Non-empty student-created method that is placed in another method.

0 = not present

1 = non-empty student-created method found within any method

User-Usability. Instructions provided for most functionality; player can engage in the game.

0 = no instructions are given

1 = most functionality described and is enough for the player to engage in game play

Programmer-Usability. Renaming objects with other than default names.

0 = not present; no renaming. She/He Builder characters are coded at "0."

1 = one or more objects are renamed; includes Dummy names

Parameters. Setting optional parameters or parameters on student-created methods such as font size, color, as seen by, and so forth, for methods. Do not count any changes in the duration parameter.

0 = no parameters set other than duration

1 = setting of optional parameters or parameters on student-created methods

Nested If/Else. The use of this is one way to implement a GUESSING mechanic.

0 = not present; no nesting of if/else statements. Program may contain if/else statements, but none are nested.

1 = one or more instances of nested if/else statements; on executable or not program paths

Repetition. To represent looping through program statements or events.

0 = not present

1 = uses loop or while statement, use of function on the looping statement must be found but can be just TRUE

Dialog Box. Player is asked for input, read in using a function, and program uses input.

0 = not present

1 = includes some programming but does not execute successfully

2 = all pieces present and execute successfully

Vehicle. The vehicle object and object attached to it move in unison.

0 = not present

1 = includes some programming but does not execute successfully

2 = all pieces present and execute successfully

Collision. Proximity function is used to trigger an action depending on the distance that one object is from another.

0 = not present

1 = present but does not execute successfully

2 = all pieces present and execute successfully

Camera Control. Changing the view according to movement or player input within one scene. If the player moved out of one scene, code as Pattern Scene Change.

0 = no camera setting or changing of views

1 = views set in methods or events or with Dummies (Alice construct) but not reached during execution

2 = created, used, and executes successfully

Scene Change. Programming movement to and from different scenes. The camera is used to move to another location where a completely different scene resides. Can be done using Dummies or using a process of fade out, make objects invisible, fade in, and make other objects visible.

0 = not present

1 = created but executed unsuccessfully or not used in game play

2 = created, used, and executes successfully

List Processing. List variables are created and used with Alice programming constructs that can manipulate these list variables.

0 = not present

1 = list variables created as type object but not used

2 = list variables are used with For all in order (or For all one at a time) or For all together (or For all every together)

Counter. Integer variable created, initialized, the variable's value is incremented or decremented based on the game state, and if the variable hits the threshold value, it triggers additional action.

0 = not present

1 = variable created and not used

2 = variable created and used incorrectly

3 = created, used, and executes successfully by changing the state of the game upon reaching a threshold

Timer. Integer variable created and initialized, the variable's value changed as time passes, and if variable hits the threshold value, it triggers additional action.

- 0 = not present
- 1 = variable created and not used
- 2 = variable created and used incorrectly
- 3 = created, used, and executes successfully by changing the state of the game upon reaching a threshold

Expand Obj Method. Student-created method to give additional functionality to an object. Method need not be called for execution during game play.

- 0 = not present
- 1 = method manipulating a single object, organized anywhere other than the corresponding object's level
- 2 = method manipulating a single object (can be the world), created at the corresponding object's level, or a single method controlling many instances of a single class of objects, organized at any level
- 3 = created with a parameter at either the world or object level

Game Mechanics

Avoidance. Player's token avoids stationary or moving objects or areas that provide negative feedback with player proximity.

- Feedback is required (e.g., low-level feedback for when the player hits an obstacle).
- Obstacles that DO NOT provide feedback when approached (either explicit or intentional, but not complete—e.g., incomplete collision detection Event) are to be treated as visual barriers and coded for NAVIGATION, not AVOIDANCE.

- 0 = not present
- 1 = present but does not work properly, often because the two objects are attached through vehicles, same key/mouse controls, and so forth
- 2 = present and works! Possible to hit/be hit by an obstacle and to avoid completely. Instructions are not required if proximity to the obstacle triggers negative feedback OR changes the game state (i.e., player needs to know that he or she must avoid obstacles but may only be known by losing the first time through game play)

Collecting. Player attempts to accumulate or gain ownership of objects. Player collects objects by clicking on or proximity to objects.

- Includes when instructions are "shoot" but no projectile is present; if a projectile is present, then SHOOTING.
- No order to the collecting; if ordering is required, then NAVIGATION.
- Can code for both HIT MOVING TARGET and COLLECTING.
- Collecting one object generally does not count, unless:
 - The intent from instructions was to have multiple objects collected but only one object collection was successfully implemented, or
 - Shooting instructions are given but no projectile and only one target object.
- Feedback is required—either state change of collected objects or game state change.

- 0 = not present
- 1 = present but does not work properly
- 2 = present with feedback

Exploration. Player moves an object or camera or both, in free roaming manner, not along a designated path, to find known or unknown objects beyond a player's initial range of view.

- “Find and click” games that are free roaming, not along a designated path, are coded as EXPLORATION only, not HIDDEN OBJECTS or NAVIGATION.
- “Find and click” games that involve going along a designated path are coded as HIDDEN OBJECTS and NAVIGATION, not EXPLORATION.
- Instructions are required, but feedback is optional.
- IF a player has to explore beyond the initial range of view to find a target object and
 - Guess something about multiple objects = EXPLORATION and GUESSING.
 - Click on something moving = EXPLORATION and HIT MOVING TARGET.
- See the RACING mechanic game for other differences.
 - Finding the finish line $\neq$ EXPLORATION, unless implied as part of the challenge.
 - Following the designated track $\neq$ EXPLORATION but RACING.

0 = not present

1 = no instructions present

1 = goal mentioned in instructions but the player cannot move to find an object or the target object does not exist in the world, or the player choosing correct object from the incorrect objects but there is no feedback saying that it is correct

2 = goal mentioned in instructions to find object(s), possible to find them, and the player knows when they are found

Guessing. Player answers questions via clicking, typing, moving. Code any mechanic or pattern used to accept player input. Feedback needed for a player to learn if the answer is correct.

0 = not present

1 = present but does not work properly

2 = present and works

Hidden Objects. Player searches known objects that are either hidden beyond view, along a navigational path, or “hidden in plain sight,” using visual confusion.

- Objects in visual confusion code as HIDDEN OBJECTS.
 - Objects hidden among near identical objects (e.g., “Which one does not match” games)—if objects are hidden among all identical objects, code instead as GUESSING.
 - Objects that are identical or not and hidden within a cluttered, random, visually chaotic environment (e.g., “Eye Spy” or “Where’s Waldo” games).
 - Objects that are identical or not and are partially hidden or covered.
- “Find and click” games = HIDDEN OBJECTS + NAVIGATION or EXPLORATION only.
 - “Find and click” where the target objects are arranged along a navigational path = HIDDEN OBJECTS and NAVIGATION.
 - “Find and click” where player is roaming freely, on a non-designated path, looking for known or unknown target objects = EXPLORATION only, not HIDDEN OBJECTS.
- Difference between EXPLORATION and HIDDEN OBJECTS:
 - EXPLORATION—objects are hidden beyond view.
 - HIDDEN OBJECTS—objects are hidden beyond view and along a navigational path.

0 = not present

1 = present but does not work properly

2 = present and works

Hit Moving Target. Player attempts to move a token to a moving target object triggering proximity function or player clicks on a moving target.

- Target need not be constantly moving.
- Double-coding: if there is a change of state of the target when approached or clicked on, code as both HIT MOVING TARGET and COLLECTING.
- Feedback is required. Can be a change of state of the target or a change of the game state.

0 = not present

1 = present but does not work properly—either no feedback or hit of moving target is not possible

2 = present and works. Behavior possible; feedback present.

Infinite Game Play. There is no explicit end, such as with Farmville or Sims. Do not code this category if the game is not complete and it is not clear what the intention was. In Alice, most games continue because the programmer has not cleared the scene using fadeout, for example, but this would not be coded as infinite game play.

0 = not present

1 = present but does not work properly

2 = present and works

Interactive. Player can interact with a game by either typing text or clicking the mouse, arrow key, number or letter keys, return, or space bar.

0 = not present

1 = present and works somewhat—insufficient or incorrect instructions for functionality

2 = present and instructions are given for at least most of the functionality

Levels. Includes more than one level where the player moves between levels, or scenes, by gathering points or other attributes. The levels do not require any difference in difficulty of tasks.

0 = not present

1 = intent stated and an attempt made, but code does not really work

2 = present and works

Navigation. Player moves a token that is an object or camera or both from “Area(s) A” to “Area(s) B.” Player must know the location(s) of end destination(s) (may be within the initial view or at the end of a given track of movement).

- Barriers optional: if present and provide negative feedback, count for AVOIDANCE.
- Designated paths optional and may be stationary, moving, or appear to be moving with visible barriers. Designated path does not have to include a target end point.
- Feedback is optional.
- Double-coding examples:
 - EXPLORATION, AVOIDANCE, COLLECTING, and HIT MOVING TARGET games would not be counted additionally for NAVIGATION unless they also have a designated path to follow as the player is “exploring,” “avoiding,” “collecting,” or “hitting moving target.”
 - “Find and click” games that use free roaming movement are coded as EXPLORATION, not HIDDEN OBJECTS and not NAVIGATION.
 - If the player gets through path and finds something, then code as NAVIGATION and HIDDEN OBJECTS.

- If there is change or elimination of objects along the navigational path, then code as NAVIGATION only, not COLLECTING.
- If avoiding objects because of negative feedback, code as AVOIDANCE. If the player must get from one location to another specific location while avoiding objects rather than just getting “away” from the obstacles, code as both AVOIDANCE and NAVIGATION.
- If navigating around visual barriers toward moving objects, code as both HIT MOVING TARGET and NAVIGATION.

0 = not present

1 = present but does not work properly. Player required to get somewhere because of indication of intention present but cannot move token.

2 = present and works.

Puzzle. Player organizes pieces to make order out of chaos/disorder.

0 = not present

1 = present but does not work properly.

2 = present and works.

Racing. Player moves an object across some kind of “finish line” or an object moves faster than other objects. Needs to include another competitor or a timed challenge.

- Feedback is required—counts as 2 even if only one object/player can trigger change in the game state; enough mastery is assumed if the student can implement one player.
- Visual barriers to movement that do not provide negative feedback if touched are found along the racetrack = RACING and NAVIGATION.
- Obstacles to movement that provide negative feedback along the racetrack = RACING and AVOIDANCE.

0 = not present

1 = present but does not work properly.

2 = present and works

Shooting. Player is shooting at things. Actual projectile must be present.

0 = not present

1 = present but does not work properly

2 = present and works

Timed Challenge. There is a timer present or mention of a timer, and that game challenge must be completed in a specified time. Indication of intention for a timed challenge is required in instructions/dialog, visual representation of a timer, or target objects leave the world by sinking below ground or moving off the screen.

0 = not present

1 = only intention is mentioned or only parts of it work

2 = present and the game “state” changes when time runs out. Mentioning a time is not required if visual representation of a timer is present, the timer runs, and the game state changes when the timer runs out.

ACKNOWLEDGMENTS

Thanks go to Huimee Sanchez and David M. Torres for coding the students’ game programs for the presence of programming constructs, patterns, and game mechanics. Thanks also go to Yethzell Díaz, Justen Time, Luci Basor, and Sara Shimel, who were the teachers of the afterschool program.

REFERENCES

- [1] Joel C. Adams and Andrew R. Webster. 2012. What do students learn about programming from game, music video, and storytelling projects? In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 643–648.
- [2] Ashok Basawapatna, Kyu Han Koh, Alexander Repenning, David C. Webb, and Krista Sekeres Marshall. 2011. Recognizing computational thinking patterns. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 245–250.
- [3] Ashok R. Basawapatna, Kyu Han Koh, and Alexander Repenning. 2010. Using scalable game design to teach computer science from middle school to graduate school. In *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, 224–228.
- [4] Karen Brennan and Mitchel Resnick. 2012. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 Annual Meeting of the American Educational Research Association*, Vol. 1. 25.
- [5] Shannon Campe, L. Werner, and Jill Denner. 2005. Information technology fluency for middle school girls. In *Proceedings of the 8th IASTED International Conference Computers and Advanced Technology in Education*.
- [6] Stephen Cooper, Lance C. Pérez, and Daphne Rainey. 2010. K–12 computational learning. *Communications of the ACM* 53, 11 (2010), 27–29.
- [7] Wanda P. Dann, Stephen Cooper, and Randy Pausch. 2008. *Learning to Program with Alice*. Prentice Hall.
- [8] Jill Denner. 2007. The Girls Creating Games program: An innovative approach to integrating technology into middle school. *Meridian: A Middle School Computer Technologies Journal* 1, 10 (2007). Available at <http://www.ncsu.edu/meridian/win2007/girlgaming/index.htm>.
- [9] Jill Denner. 2011. What predicts middle school girls interest in computing? *International Journal of Gender, Science and Technology* 3, 1 (2011), 54–69.
- [10] Jill Denner, Steve Bean, and Jacob Martinez. 2009. The Girl Game Company: Engaging Latina girls in information technology. *Afterschool Matters* 8 (2009), 26–35.
- [11] Jill Denner, Shannon Campe, and Linda Werner. 2019. Does computer game design and programming benefit children? A meta-synthesis of research. *ACM Transactions on Computing Education* 3 (2019), 1–35.
- [12] Jill Denner, Linda Werner, Shannon Campe, and Eloy Ortiz. 2014. Pair programming: Under what conditions is it advantageous for middle school students? *Journal of Research on Technology in Education* 46, 3 (2014), 277–296.
- [13] Jill Denner, Linda Werner, and Eloy Ortiz. 2012. Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts? *Computers & Education* 58, 1 (2012), 240–249.
- [14] Allison Elliott Tew and Brian Dorn. 2013. The case for validated tools in computer science education research. *Computer* 46, 9 (2013), 60–66.
- [15] Deborah A. Fields, Kristin A. Searle, Yasmin B. Kafai, and Hannah S. Min. 2012. Debuggems to assess student learning in e-textiles. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 699.
- [16] Diana Franklin, Phillip Conrad, Bryce Boe, Katy Nilsen, Charlotte Hill, Michelle Len, Greg Dreschler, et al. 2013. Assessment of computer science learning in a scratch-based outreach program. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 371–376.
- [17] Shuchi Grover, Stephen Cooper, and Roy Pea. 2014. Assessing computational learning in K-12. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, 57–62.
- [18] Shuchi Grover and Roy Pea. 2013. Computational thinking in K–12: A review of the state of the field. *Educational Researcher* 42, 1 (2013), 38–43.
- [19] Robin Hunnicke, Marc LeBlanc, and Robert Zubek. 2004. MDA: A formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, Vol. 4. 1722.
- [20] Yasmin B. Kafai, Eunkyoun Lee, Kristin Searle, Deborah Fields, Eliot Kaplan, and Debora Lui. 2014. A crafts-oriented approach to computing in high school: Introducing computational concepts, practices, and perspectives with electronic textiles. *ACM Transactions on Computing Education* 14, 1 (2014), 1.
- [21] Yasmin B. Kafai, Kristin Searle, Eliot Kaplan, Deborah Fields, Eunkyoun Lee, and Debora Lui. 2013. Cupcake cushions, scooby doo shirts, and soft boomboxes: e-Textiles in high school to promote computational concepts, practices, and perceptions. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 311–316.
- [22] Kyu Han Koh, Ashok Basawapatna, Hilarie Nickerson, and Alexander Repenning. 2014. Real time assessment of computational thinking. In *Proceedings of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'14)*. IEEE, Los Alamitos, CA, 49–52.

- [23] Kyu Han Koh, Hilarie Nickerson, Ashok Basawapatna, and Alexander Repenning. 2014. Early validation of computational thinking pattern analysis. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education (ITiCSE'14)*. ACM, New York, NY, 213–218.
- [24] Irene Lee, Fred Martin, Jill Denner, Bob Coulter, Walter Allan, Jeri Erickson, Joyce Malyn-Smith, and Linda Werner. 2011. Computational thinking for youth in practice. *ACM Inroads* 2, 1 (2011), 32–37.
- [25] Yvonna S. Lincoln and Egon G. Guba. 1985. *Naturalistic Inquiry*. Vol. 75. Sage.
- [26] Marcia C. Linn. 1985. The cognitive consequences of programming instruction in classrooms. *Educational Researcher* 14, 5 (1985), 14–29.
- [27] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*. ACM, New York, NY, 125–180.
- [28] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2013. Learning computer science concepts with scratch. *Computer Science Education* 23, 3 (2013), 239–264.
- [29] Matthew B. Miles and Michael Huberman. 1994. *Qualitative Data Analysis: An Expanded Sourcebook*. Sage.
- [30] Anthony J. Onwuegbuzie and Nancy L. Leech. 2007. Validity and qualitative research: An oxymoron? *Quality & Quantity* 41, 2 (2007), 233–249.
- [31] Sofia Papavlasopoulou, Michail N. Giannakos, and Letizia Jaccheri. 2018. Discovering children's competences in coding through the analysis of Scratch projects. In *Proceedings of the 2018 IEEE Global Engineering Education Conference (EDUCON'18)*. IEEE, Los Alamitos, CA, 1127–1133.
- [32] Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books.
- [33] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education* 13, 2 (2003), 137–172.
- [34] Jeremy Scott. 2013. The Royal Society of Edinburgh/British Computer Society computer science exemplification project. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, 315–315.
- [35] Chris Stephenson. 2016. *What is Computational Thinking?* Retrieved January 28, 2020 from <https://www.youtube.com/watch?v=FuN6g8NmuHc>.
- [36] Kathryn T. Stolee and Teale Fristoe. 2011. Expressing computer science concepts through Kodu game lab. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 99–104.
- [37] Renata Tesch. 1990. *Qualitative Research: Analysis Types and Software Tools*. Psychology Press.
- [38] Linda Werner, Jill Denner, and Shannon Campe. 2015. Children programming games: A strategy for measuring computational learning. *ACM Transactions on Computing Education* 14, 4 (2015), 24.
- [39] Linda Werner, Jill Denner, Shannon Campe, and Damon Chizuru Kawamoto. 2012. The fairy performance assessment: Measuring computational thinking in middle school. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 215–220.
- [40] Linda L. Werner, Shannon Campe, and Jill Denner. 2005. Middle school girls + games programming = information technology fluency. In *Proceedings of the 6th Conference on Information Technology Education*. ACM, New York, NY, 301–305.
- [41] Iris Zur-Bargury, Bazil Pârva, and Dvir Lanzberg. 2013. A nationwide exam as a tool for improving a new curriculum. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, 267–272.

Received November 2018; revised December 2019; accepted January 2020